AutoMon: Automatic Distributed Monitoring for Arbitrary Multivariate Functions

Hadar Sivan
Technion – Israel Institute of Technology
Haifa, Israel
hadarsivan@cs.technion.ac.il

Moshe Gabel
University of Toronto
Toronto, Canada
mgabel@cs.toronto.edu

Assaf Schuster
Technion – Israel Institute of Technology
Haifa, Israel
assaf@cs.technion.ac.il

ABSTRACT

Approaches for evaluating functions over distributed data streams are increasingly important as data sources become more geographically distributed. However, existing methodologies are limited to small classes of functions, requiring non-trivial effort and substantial mathematical sophistication to tailor them to new functions.

In this work we present AutoMon, the first general solution to this problem. AutoMon enables automatic, communication-efficient distributed monitoring of arbitrary functions. Given source code that computes a function from centralized data, the AutoMon algorithm approximates the function over the aggregate of distributed data streams, without centralizing data updates.

Our evaluation shows that AutoMon sends the same number or fewer messages as state-of-the-art techniques when monitoring specific functions for which a distributed, hand-crafted solution is known. AutoMon, however, is a lot more powerful. It automatically generates a communication-efficient distributed monitoring solution for arbitrary functions, e.g., monitoring deep neural networks inference tasks for which no non-trivial solution is known.

CCS CONCEPTS

• **Theory of computation** → **Distributed algorithms**; *Streaming, sublinear and near linear time algorithms*; *Streaming models*; • **Information systems** → *Data streams*; *Stream management*.

KEYWORDS

distributed streams; functional monitoring; approximation

ACM Reference Format:
Hadar Sivan, Moshe Gabel, and Assaf Schuster. 2022. AutoMon: Automatic Distributed Monitoring for Arbitrary Multivariate Functions. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD '22), June 12–17, 2022, Philadelphia, PA, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3514221.3517866

SIGMOD '22, June 12–17, 2022, Philadelphia, PA, USA.

ACM ISBN 978-1-4503-9249-5/22/06…$15.00
https://doi.org/10.1145/3514221.3517866

1 INTRODUCTION

Consider the problem of determining whether a network is currently under attack using the aggregate of local statistics from multiple routers [18]. State-of-the-art approaches use machine learning models trained to detect attacks or find outliers given network metrics [24, 68]. For example, we might detect attacks by continuously evaluating the output of a trained neural network:

$$f_{nn}(\bar{x}) = W_3 \cdot \tanh\left(W_2 \cdot \tanh\left(W_1 \cdot \bar{x} + b_1\right) + b_2\right) + b_3 \ ,$$

where the input vector $\bar{x} = \frac{1}{k}\sum_{i=1}^{k} x^i$ is the average of k *dynamic* router metric vectors $x^i \in \mathbb{R}^d$ that change over time, the matrices W_i and vectors b_i are the weights of the neural network, and tanh is applied element-wise. In a centralized setting, computing f_{nn} is a straightforward task for the average software developer:

```
from numpy import tanh
def f_nn(x, W1, b1, W2, b2, W3, b3):
   return W3 @ tanh(W2 @ tanh(W1 @ x + b1) + b2) + b3
```

When the vectors x^i change, we can simply recompute $f_{nn}(\bar{x})$ as needed, assuming sufficient computational power.

However, the problem becomes much more difficult once the vectors x^i are distributed, even if we assume sufficient computational power and allow an approximation of f_{nn} rather than computing the exact value. The root problem is that f_{nn} is highly non-linear, making it difficult to understand how it will be affected by a change in x^i. Although, in theory, we could centralize all data updates, this can be infeasible or undesirable in a geographically-distributed environment since communication incurs power and bandwidth costs at the origin nodes of x^i [5, 34, 61]. Returning to our previous example, continuously sending statistics from the routers may use up too much network bandwidth [16, 22, 27, 44]. Yet, only sending periodic updates risks missing or delaying the detection of an attack [8, 22]. Resource-limited data sources in mobile computing and Internet-of-Things have further heightened the need for communication-efficient distributed data stream monitoring [53, 58, 59], since the wide geographical distribution of data sources coupled with resource limitation prohibits centralizing all data updates [25]. Other scenarios where complex decisions must be made based on global data include battery-powered wireless sensor networks and edge computing. Distributed monitoring in these settings is hotly studied since centralizing data is impractical due to battery limitations and limited links [5, 35, 48].

The task of continuously evaluating a multivariate function from an aggregate of several data vectors that change over time is a variant of *distributed functional monitoring* in the *continuous distributed monitoring model* [14, 15, 65]. Any general approach for this task

faces two main challenges. First, given a function f, how can we maintain an estimate of $f(\bar{x})$ while avoiding the need to send all data updates of x^i? Second, how do we make it accessible to an average software developer, who may not have the mathematical skills required to tailor the approach to a specific problem?

Most existing work on general distributed functional monitoring focuses on a single aspect of the problem. For example, while Geometric Monitoring [40, 57] and Convex Bound [41] have been used to compute a wide variety of functions such as variance [23], spectral gap [67], skylines [51], and least-squares regression [21], applying these approaches for each new function requires in-depth mathematical analysis. Conversely, Universal Sketching techniques [12] provide multiplicative approximation that are easy to use, but are limited to a subset of monotone functions of item counts (i.e., $\bar{x}$ must be a frequency vector). Distributed data analysis frameworks [13, 29, 45, 48, 66] require no math to use, but can only optimize a limited set of primitives.[1] Neither approach is suitable for low-communication monitoring of more complex functions such as neural networks (i.e., f_{nn}), which can be more accurate in detecting outliers, failures, and network attacks [24, 33].

Our Contributions: We describe AutoMon, short for Automatic Monitoring, an algorithmic building block that enables automatic distributed functional monitoring for difficult functions for which no hand-crafted solution is known while addressing both of the above challenges. Given a source code snippet for any function f of the aggregate vector $\bar{x}$ and the desired approximation error, AutoMon *automatically* implements a communication-efficient approximation for f over multiple nodes, each with its own dynamic data vector. In particular, we make the following contributions:

- A novel communication-efficient scheme for monitoring arbitrary functions of the global aggregated vector. Given a function's source code, we leverage automatic differentiation [7], numerical optimization, and the Geometric Monitoring protocol [3, 40] to derive local constraints that the nodes can check locally, avoiding communication when changes to local data are too small to violate the approximation bounds.
- An extensive evaluation on synthetic and real-world datasets, and on a range of different functions, including KL-divergence, inner product, and neural networks (DNN). AutoMon provides a superior error-communication tradeoff to existing methods. For example, on a DNN approximation task – for which no efficient distributed approximation is known – AutoMon reduces number of messages and bandwidth usage by up to two orders of magnitude, compared to centralization.
- A prototype open source implementation of AutoMon. Our prototype library provides an unobtrusive API designed to facilitate development of stand-alone distributed applications (e.g., neural networks over data streams) and components of data analysis frameworks (e.g., efficient implementation of custom operators in stream processing engines [19]).

To the best of our knowledge, AutoMon is the first truly automatic distributed functional-monitoring scheme that supports a wide range of functions defined on arbitrary data, and works directly from the source code of the function to compute without manual mathematical analysis.

AutoMon is available as an open source project on GitHub: https://github.com/hsivan/automon.

2 BACKGROUND

Consider a distributed system with a single coordinator node and n nodes, where each node i holds a *dynamic* local data vector x^i computed from its local data stream; x^i changes arbitrarily over time and nodes only communicate with the coordinator[2] [14]. When clear from context, we omit i and use x to denote a local vector.

Let f be an arbitrary real multivariate function $f : \mathbb{R}^d \to \mathbb{R}$ of the average vector of local data $\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x^i$. Given f expressed as code in a high-level language (e.g., Python or C++) and an approximation error bound ϵ, we wish to maintain an ϵ-approximation of $f(\bar{x})$, and do so with minimal communication. This is a variation of the *distributed functional monitoring* task [65].[3] The difference is that we aim to support arbitrary functions expressed as programs.

Note that we can use threshold monitoring to obtain such an approximation: given a *reference point* x_0, which is the value of $\bar{x}$ at some point in time, we can provide an additive approximation by setting two thresholds, L and U, to be $f(x_0) \pm \epsilon$ and require that $L \leq f(\bar{x}) \leq U$; to obtain a multiplicative approximation of $f(\bar{x})$ we set L and U to $(1 \pm \epsilon)f(x_0)$. As long as $L \leq f(\bar{x}) \leq U$, $f(x_0)$ is an ϵ-approximation of $f(\bar{x})$; if not, we update x_0, U, and L.

Our strategy is to automatically compute local constraints on the local data of each node. These constraints should be: (1) *correct* – as long as all local constraints hold, the *global condition* $L \leq f(\bar{x}) \leq U$ is guaranteed to hold; (2) *efficient* – the number of times the local constraints are violated is minimal, resulting in less communication; and (3) *automatic* – can be computed using the source code of f, without requiring mathematical insight or developer effort.

Finding local constraints that are correct, efficient, and automatic is a challenging task, and even more so for an arbitrary function. We now briefly review the necessary background. We describe our method in §3 and its implementation details in §3.8.

DC Decomposition: We use a *DC decomposition* [2] of f to derive local constraints that provide correctness. DC decomposition is a representation of a function as the difference of two convex or concave functions. We use the term *convex difference* for the representation of a function as a difference of two convex functions, and the term *concave difference* for the representation of a function as a difference of two concave functions. Hence, if $\breve{g}(x)$ and $\breve{h}(x)$ are convex functions such that $f(x) = \breve{g}(x) - \breve{h}(x)$, we can rewrite the global condition $L \leq f(\bar{x}) \leq U$ as $\breve{g}(\bar{x}) \leq \breve{h}(\bar{x}) + U$ and $\breve{h}(\bar{x}) \leq \breve{g}(\bar{x}) - L$. Similarly, if $\hat{g}(x)$ and $\hat{h}(x)$ are concave functions such that $f(x) = \hat{g}(x) - \hat{h}(x)$, then we can rewrite the global condition as $\hat{h}(\bar{x}) \geq \hat{g}(\bar{x}) - U$ and $\hat{g}(\bar{x}) \geq \hat{h}(\bar{x}) + L$.

[1] For example, SQL-based approaches are limited to grouping, ordering, count, sum, average, and so on [64], while stream processing frameworks are similarly limited to built-in aggregates, windowing, and maps [13]. Such approaches only express a small part of the space of possible computations [46].

[2] These assumptions are for clarity and are not central to our design. First, communication need not be direct – we assume an underlying message passing protocol or distributed control plane. Similarly, the coordinator holds little state and need not be unique. AutoMon can be implemented using converge-casting [9], hierarchical violation resolution [37], or consensus protocols [32, 49].

[3] Though not immediately obvious, a huge variety of computations can be expressed as $f(\bar{x})$ by augmenting the local vectors x^i [21, 22, 25, 37, 40, 41, 51].

Automatic Differentiation: We use *automatic differentiation* (AD) to find a DC decomposition of a function. AD is a general method for taking a function specified by a computer program and automatically constructing a procedure to compute the derivatives of that function [7]. Unlike symbolic differentiation, which outputs a closed-form symbolic formula for the derivative, AD outputs a computational graph that can be evaluated efficiently at runtime for specific inputs. This means AD can be applied to standard numeric program code, making it suitable for our purposes. Upon initialization, AD explicitly constructs the computational graph of the function, and repeatedly applies the chain rule to this graph to compute the function's derivatives of arbitrary order. This results in a procedure for computing derivatives.

Geometric Monitoring Protocol: AutoMon adopts the geometric monitoring (GM) protocol for continuous threshold monitoring in a distributed system, which has been widely adopted by distributed monitoring methods [21–23]. §3 provides a detailed description of the AutoMon protocol; what follows is a brief summary of the generic GM protocol.

The GM protocol comprises two basic parts: the coordinator algorithm and the node algorithm. Each node receives local data and updates its dynamic local vector x. A node is responsible for monitoring the local constraints, reporting violation of these constraints to the coordinator, and receiving updated constraints from the coordinator. The correctness of local constraints guarantees that if all nodes have no reported violation, the global condition is maintained. The coordinator is responsible for resolving violations of the local constraints by distributing updated local constraints to nodes or approximation bounds L, U, as needed. When the coordinator is notified of local violations, it collects the local vectors from the nodes and, if needed, also updates the reference point x_0 to the average vector $\bar{x}$ at the time. After the data centralization, the coordinator computes new local constraints that resolve the violations and synchronizes the nodes with the new constraints.

Let $\mathcal{D}$ denote the domain where $f(x)$ is defined. GM defines an *admissible region* $\mathcal{A}$ as the subset of $\mathcal{D}$, where $L \leq f(\bar{x}) \leq U$. Given a local constraint, GM also defines a *safe zone*: the subset of $\mathcal{D}$ in which the local constraints of a node are satisfied. If the safe zone is convex and is a subset of the admissible region, the GM protocol guarantees that the approximation bound defined by the thresholds (i.e., the global condition $L \leq f(\bar{x}) \leq U$) is maintained [22, 40]. When the resulting safe zone is not convex, this gives rise to the possibility of *missed violations*. This occurs when the global condition is not maintained ($\bar{x}$ is outside the admissible region), yet there is no violation in any of the local constraints.

Automatically Deriving Local Constraints: The GM protocol itself is conceptually simple, since much of the "heavy lifting" is done by the local constraint required by the protocol. Indeed, the convexity of the safe zone is a key non-trivial requirement on the local constraint. Previous work relied on manual analysis and researcher expertise to find local constraint for specific functions [21–23, 51, 67]. However, we are faced with the greater challenge of doing so automatically for arbitrary functions expressed as code, without requiring in-depth mathematical analysis of each function. In the next section, we describe how we overcome this challenge using automatic differentiation and numerical optimization.

3 AUTOMATIC DISTRIBUTED MONITORING

We aim to provide an automatic method for distributed monitoring of arbitrary functions of the global aggregate $\bar{x}$. Given a function specified by a computer program, we automatically generate a communication-efficient scheme to monitor this function.

We first describe ADCD (for *Automatic DC Decomposition*), the automatic local constraint technique that lies at the heart of AutoMon. ADCD uses automatic differentiation and numerical optimization to derive local constraints for arbitrary functions. We describe two variants of ADCD, one for general functions (§3.1) and the other for functions with constant Hessian (§3.2). ADCD detects the type of the function and uses the best ADCD variant accordingly to provide a DC decomposition, which it then converts to a GM-style local constraint (§3.3) that can be plugged-in to the GM protocol. We also explore how the type of DC decomposition (convex or concave) affects the quality of the derived constraints, and propose a heuristic for choosing between convex difference and concave difference (§3.4).

We then describe how AutoMon combines ADCD with the GM protocol to do functional monitoring (§3.5). Additionally, we consider a novel aspect of the problem: the impact of limiting the monitoring to a small part of the domain in a neighborhood around the reference point, using local constraints that are customized to this neighborhood (§3.6). Finally, we discuss correctness guarantees (§3.7) and implementation considerations (§3.8).

3.1 ADCD by Extreme Eigenvalue (ADCD-X)

The following lemma shows how to obtain a DC decomposition of a twice differential function $f(x)$. Recall a function f is convex if and only if its Hessian H is positive semidefinite (denoted $H \succeq 0$), i.e., its smallest eigenvalue is non-negative. Conversely, f is concave if its largest eigenvalue is non-positive, $H \preceq 0$. The idea behind the lemma is that f can be "made convex" by adding another function such that the Hessian is positive semidefinite (PSD). The added function must be convex and the difference between the altered function and the added function is the required DC decomposition. The added function construction is based on the extreme eigenvalues of the Hessian of the function. Note, the lemma is defined over some set $\mathcal{S}$, which can be the full domain $\mathcal{D}$ or a subset of it.

Lemma 1. *Let $f(x)$ be a twice differentiable function with domain $\mathcal{D}$, and let $\mathcal{S} \subseteq \mathcal{D}$ be a subset of the domain. Let $\lambda_{\min}$ and $\lambda_{\max}$ be the smallest and largest eigenvalues of the Hessian $H(x)$ of $f(x)$ where $x \in \mathcal{S}$, and define $\lambda^-_{\min} := \min\{0, \lambda_{\min}\}$ and $\lambda^+_{\max} := \max\{0, \lambda_{\max}\}$.*

Then (1) is a convex difference of $f(x)$ over $\mathcal{S}$:

$$f(x) = \underbrace{f(x) + \frac{1}{2}\left|\lambda^-_{\min}\right|\|x - x_0\|^2}_{\text{convex } \check{g}(x)} - \underbrace{\frac{1}{2}\left|\lambda^-_{\min}\right|\|x - x_0\|^2}_{\text{convex } \check{h}(x)} \tag{1}$$

and (2) is a concave difference of $f(x)$ over $\mathcal{S}$:

$$f(x) = \underbrace{f(x) - \frac{1}{2}\lambda^+_{\max}\|x - x_0\|^2}_{\text{concave } \hat{g}(x)} - \underbrace{\frac{-1}{2}\lambda^+_{\max}\|x - x_0\|^2}_{\text{concave } \hat{h}(x)}. \tag{2}$$

The proof of Lemma 1 uses the fact that $\lambda^-_{\min} \leq \lambda_{\min}$ to show that Hessians of $\check{g}$ and $\check{h}$ are PSD, which implies $\check{g}$ and $\check{h}$ are convex, and

similarly show that $\hat{g}$ and $\hat{h}$ are concave. The full proof is omitted due to space limitation.

Lemma 1 shows how to construct a DC decomposition if we are given the extreme eigenvalues.[4] Alas, finding the extreme eigenvalues of a general function, with an x-dependent Hessian matrix, is a difficult task [50]. The Hessian $H(x)$ is a function of x, and therefore its eigenvalues are a function of x. Finding the x in $\mathcal{S}$ that obtains the minimal or maximal eigenvalue is not trivial. Therefore, instead of finding the extreme eigenvalues analytically, we use numerical techniques to solve this problem.

Using Automatic Differentiation: Using Lemma 1 requires obtaining $H(x)$ and finding $x', x'' \in \mathcal{S}$ that obtain $\lambda_{\min}$ and $\lambda_{\max}$. Our key insight here is that we do not need an analytic solution for $\lambda_{\min}$ and $\lambda_{\max}$, nor a symbolic expression of $H(x)$, but rather a way to evaluate H for specific points in $\mathcal{S}$. Since we are given an arbitrary $f(x)$ in the form of a short program, AD enables this automatic evaluation of H in any $x \in \mathcal{S}$. By evaluating H using AD at multiple points in $\mathcal{S}$, and computing the extreme eigenvalues of each such Hessian, we can find the global minimum and maximum $\lambda_{\min}$ and $\lambda_{\max}$. Instead of evaluating H at random points, we define and solve an optimization problem, which is a more robust and efficient method to find these extreme values.

Another advantage of using AD is that it allows us to apply Lemma 1 to functions that are not strictly twice-differentiable. As demonstrated by the DNN with ReLU activation in §4, AD allows us to surpass this limitation as long as the function is continuous.

Finding the Eigenvalues: After having $H(x)$ computed by AD, and the ability to evaluate it at every $x \in \mathcal{S}$, we can now use it to find the extreme eigenvalues. For this $H(x)$, we define two functions, $\lambda_{\min}(H(x))$ and $\lambda_{\max}(H(x))$, which yield the minimal and maximal eigenvalues of the Hessian, respectively, at a given point x. We then use numerical box-constrained optimization methods (e.g., SLSQP and L-BFGS-B) to solve two optimization problems over $\mathcal{S}$:

$$\hat{\lambda}_{\min} = \min_{x \in \mathcal{S}} \lambda_{\min}(H(x)), \quad \hat{\lambda}_{\max} = \max_{x \in \mathcal{S}} \lambda_{\max}(H(x)). \tag{3}$$

These optimization problems are not convex, and their complexity is determined by the function $f(x)$ and its Hessian. Therefore, there is no guarantee that the solution found by the optimization process is the global solution over $\mathcal{S}$. The numerical optimization algorithm could converge to a local minimum/maximum, a saddle point, or even not fully converge, as the number of iterations of the algorithm is limited and the problem could be ill-conditioned and require more iterations. Hence, $\hat{\lambda}_{\min}$ and $\hat{\lambda}_{\max}$ may be different from the true $\lambda_{\min}$ and $\lambda_{\max}$. We discuss the impact on correctness in §3.7.

3.2 ADCD by Eigendecomposition (ADCD-E)

DC decomposition, the representation of a function $f(x)$ as a difference of two convex/concave functions, is not unique. Lemma 1 in the previous section presented a specific DC decomposition of a function, in which the two functions are constructed using the eigenvalues of $f(x)$. In this section, we present ADCD-E, another DC decomposition for functions with constant Hessian. We show that this DC decomposition is superior to ADCD-X for this type of function. The decision whether to use ADCD-E or ADCD-X is done automatically by our algorithms, as it able to automatically identify the type of the function.

The main idea behind ADCD-E is to use eigendecomposition to split the Hessian of $f(x)$ into two matrices: a positive semidefinite (PSD) matrix H^+ and a negative semidefinite (NSD) matrix H^-. This can be done because H is a constant matrix. We obtain a convex difference of $f(x)$ using H^- instead of $\lambda^-_{\min}$, and a concave difference using H^+ instead of $\lambda^+_{\max}$. As before, we use automatic differentiation tools to obtain H, H^-, and H^+ .

Lemma 2. *Let $f(x)$ be a twice differentiable function with Hessian H that is not a function of x (i.e., it is a constant). Let $\lambda_1 \le \lambda_2 \le ... \le \lambda_d$ be the eigenvalues of H, and $v_1, v_2, ..., v_d$ the corresponding eigenvectors. The Hessian matrix H is a real symmetric matrix, and can therefore be decomposed as $H = Q\Lambda Q^T$, where Q is an orthonormal matrix whose columns are the eigenvectors of H, and Λ is a diagonal matrix whose entries are the eigenvalues of H.*

Let Λ^- be a diagonal matrix whose diagonal is $[\lambda_1, ..., \lambda_k, 0, ..., 0]$, and Λ^+ a diagonal matrix with diagonal $[0, ..., 0, \lambda_k + 1, ..., \lambda_d]$, where $\lambda_1, ..., \lambda_k$ are the negative eigenvalues and $\lambda_{k+1}, ..., \lambda_d$ are the non-negative eigenvalues.

Let $H^- := Q\Lambda^- Q^T$ be the NSD part of H, and $H^+ := Q\Lambda^+ Q^T$ be the PSD part. Then a convex difference of $f(x)$ is:

$$f(x) = \underbrace{f(x) - \frac{1}{2}(x - x_0)^T H^-(x - x_0)}_{convex\ \breve{g}(x)} - \underbrace{\frac{-1}{2}(x - x_0)^T H^-(x - x_0)}_{convex\ \breve{h}(x)},$$

and a concave difference of $f(x)$ is:

$$f(x) = \underbrace{f(x) - \frac{1}{2}(x - x_0)^T H^+(x - x_0)}_{concave\ \hat{g}(x)} - \underbrace{\frac{-1}{2}(x - x_0)^T H^+(x - x_0)}_{concave\ \hat{h}(x)}.$$

Proof. Because $\Lambda = \Lambda^- + \Lambda^+$, hence $H = Q\Lambda Q^T = H^- + H^+$. The Hessian of $\breve{g}(x)$ is $H - H^-$, which equals H^+. The matrix $H^+ = Q\Lambda^+ Q^T$ is PSD by construction since its eigenvalues are all non-negative from the definition of Λ^+. The Hessian of $\breve{h}(x)$ is $-H^-$, which is PSD. Hence, $\breve{g}(x)$ and $\breve{h}(x)$ are convex. The proof of concavity for $\hat{g}(x)$ and $\hat{h}(x)$ is similar. □

ADCD-E can only be applied to functions that have a constant Hessian, such as inner products or quadratic forms. For such functions, ADCD-E is superior to ADCD-X since the former results in a larger safe zone and therefore fewer local constraint violations. Intuitively, $\breve{g}_1$ is "more convex" than $\breve{g}_2$, where $\breve{g}_1$ is $\breve{g}$ from Lemma 1 and $\breve{g}_2$ is $\breve{g}$ from Lemma 2.

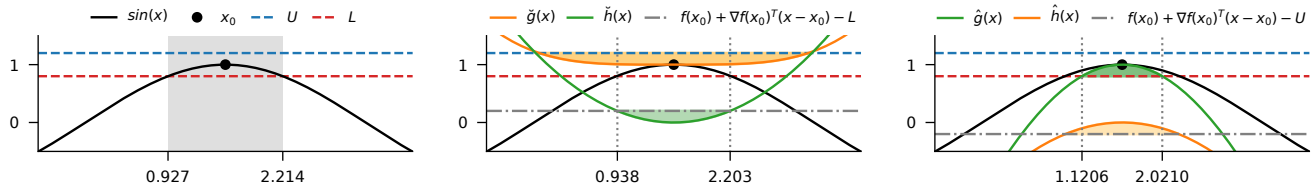
More formally, for a function with constant H, $H_{\breve{g}_1} \ge H_{\breve{g}_2}$. The proof follows from $H^- + \left|\lambda^-_{\min}\right| I \ge 0$, and therefore:

$$H_{\breve{g}_1} = H + \left|\lambda^-_{\min}\right| I = H^+ + H^- + \left|\lambda^-_{\min}\right| I \ge H^+ = H_{\breve{g}_2},$$

Further, note that $\breve{g}_1(x_0) = \breve{g}_2(x_0)$. This implies $\breve{g}_1 \ge \breve{g}_2$ for every $x \in \mathcal{D}$ (and similarly for $\breve{h}$), which means the ADCD-X safe zone is a subset of the ADCD-E safe zone.

In summary, a safe zone violation with ADCD-E implies violation with ADCD-X but a violation with ADCD-X does not necessarily imply one with ADCD-E. We can automatically detect functions with a constant Hessian by looking at the computational graph for $H(x)$ that is derived from the automatic differentiation step.

[4] An informal version of (1) appears in [41], where it is used for manual analysis of specific functions rather than automatically for general function.



(a) Admissible region is $[0.927, 2.214]$. **(b) Convex difference, safe zone is** $[0.928, 2.203]$.**(c) Concave difference, safe zone is** $[1.121, 2.202]$.

Figure 1: ADCD local constraints for $\sin(x)$ **at** $x_0 = \frac{\pi}{2}$**. Left: approximation bounds** L **and** U **and the resulting admissible region (gray span). Middle:** $\check{g}, \check{h}$ **of the convex difference from Lemma 1. The green area shows the convex set from the upper threshold condition, and the orange area shows the convex set of the lower threshold condition. The safe zone resulting from their intersection is the area between the vertical dotted lines. Right: same as middle but for** $\hat{g}$ **and** $\hat{h}$ **from the concave difference.**

3.3 From DC Decomposition to Constraints

After obtaining a DC decomposition of a function, the next step is to derive the ADCD local constraints. We now show this derivation, and provide a proof that the resulting safe zone is convex, and hence guarantee correctness.

Given a convex difference $f(x) = \check{g}(x) - \check{h}(x)$, we adopt the method of Lazerson et al. [41] to derive the ADCD local constraints for $f(x)$ using the tangent plane to $\check{g}(x)$ or $\check{h}(x)$ at x_0:

$$\check{g}(x) \leq \check{h}(x_0) + \nabla\check{h}(x_0)^T(x - x_0) + U, \tag{4a}$$

$$\check{h}(x) \leq \check{g}(x_0) + \nabla\check{g}(x_0)^T(x - x_0) - L. \tag{4b}$$

We extend this formulation for the concave difference $f(x) = \hat{g}(x) - \hat{h}(x)$, and obtain the ADCD local constraints for this difference:

$$\hat{h}(x) \geq \hat{g}(x_0) + \nabla\hat{g}(x_0)^T(x - x_0) - U, \tag{5a}$$

$$\hat{g}(x) \geq \hat{h}(x_0) + \nabla\hat{h}(x_0)^T(x - x_0) + L. \tag{5b}$$

These constraints are convex: by opening brackets and rearranging the inequality, each inequality can be written as $\psi(x) \leq C$, where ψ is a convex function and C is a constant, and the sets that satisfy such inequalities (sublevel sets) are convex [10].

For the specific convex difference, in Lemma 1 and in Lemma 2, the ADCD local constraints (4) can be simplified to:

$$\check{g}(x) \leq U, \quad \check{h}(x) \leq f(x_0) + \nabla f(x_0)^T(x - x_0) - L.$$

For the concave difference the ADCD local constraints (5) are:

$$\hat{h}(x) \geq f(x_0) + \nabla f(x_0)^T(x - x_0) - U, \quad \hat{g}(x) \geq L.$$

To get the simplified form of (4), we simply note that for $\check{g}, \check{h}$ in both Lemmas, $\check{h}(x_0) = 0$ and $\nabla\check{h}(x_0) = 0$ and, $\check{g}(x_0) = \check{f}(x_0)$ and $\nabla\check{g}(x_0) = \nabla\check{f}(x_0)$. Similarly, we can get the simplified form of (5).

Figure 1 shows an example of the ADCD local constraints derived for $\sin(x)$ at point $x_0 = \pi/2$ according to Lemma 1. Figure 1(a) shows the admissible region, while 1(b) and 1(c) show the ADCD local constraints and the resulting safe zones when using convex and concave difference representations, respectively. While both safe zones are a subset of the admissible region, they are not equivalent; We explore this in the next subsection.

3.4 Convex vs. Concave Difference

Both ADCD-X and ADCD-E provide two possible representations for a function: as a convex or as a concave difference. In some cases, a convex difference is more efficient and results in fewer safe zone violations, while in other cases the concave difference is preferable.

Consider again the example in Figure 1 showing $\sin(x)$ with the reference point $x_0 = \pi/2$. The convex difference representation in Figure 1(b) results in a wider safe zone than the concave representation in Figure 1(c). Since f near x_0 is already concave, using the concave difference results in a $\hat{g}(x)$ that is even more concave around x_0 than the original function $f(x)$. However, using the convex difference obtains a convex function $\check{g}(x)$ that is "wider" than the concave function $\hat{g}(x)$, and "wider" functions tend to obtain larger safe zones. Hence, in this case, the convex difference representation is preferable.

The curvature of $\check{g}$, $\check{h}$, $\hat{g}$, and $\hat{h}$ is determined by the eigenvalues of the Hessians of these functions, and this curvature impacts the performance of the algorithm. Therefore, we propose the *DC Heuristic* for choosing between the convex difference and concave difference, based on these eigenvalues: if

$$\lambda_{\min}\left(H_{\check{g}}(x_0)\right) + \lambda_{\min}\left(H_{\check{h}}(x_0)\right) \leq \left|\lambda_{\max}\left(H_{\hat{h}}(x_0)\right) + \lambda_{\max}\left(H_{\hat{g}}(x_0)\right)\right|$$

use the convex difference, otherwise use the concave difference.

The intuition behind this heuristic is to choose the representation whose two functions are less convex/concave near the reference point x_0. For functions with a constant Hessian, when using ADCD-E, the heuristic condition is equivalent to $|\lambda_{\min}| \leq \lambda_{\max}$.

In our preliminary experiments, this heuristic reduced safe zone violations by up to 30% when compared to using either simply the convex difference or simply the concave difference when monitoring functions such as $\sin(x)$.

3.5 The Distributed Protocol

We can now describe the protocol for AutoMon coordinator and nodes, which is based on the GM protocol; the protocol is summarized in Algorithm 1. The coordinator first collects local vectors, sets $x_0 = \bar{x}$, and updates the thresholds U and L based on $f(x_0)$ and the desired approximation. Next, the coordinator uses ADCD to derive correct convex local constraints in a *neighborhood* $\mathcal{B}$ around x_0. Finally, it distributes these local constraints to all nodes.

Let $\mathcal{B}$ be the neighborhood of size r around the reference point x_0: $\mathcal{B} = \{x : x \in [x_0 - r, x_0 + r]\}$.[5] When the neighborhood is restricted, we have two types of violations. The first type is *safe zone violation*,

[5] In practice, we also restrict the neighborhood $\mathcal{B}$ to be contained in the domain $\mathcal{D}$.

Algorithm 1 AutoMon protocol for coordinator and node.

1: **procedure** CoordinatorFullSync
2: Pull all nodes x^i and update x_0: $x_0 \leftarrow \frac{1}{n}\sum_{i=1}^{n} x^i$
3: Use $f(x_0)$ to update the thresholds L and U
4: Update the neighborhood $\mathcal{B}$ from x_0
5: Compute DC decomposition of f
6: Choose between convex difference and concave difference
7: Derive safe zone based on the chosen DC
8: Sync all nodes with the safe zone and neighborhood $\mathcal{B}$
9: **procedure** NodeDataUpdate(sample from local stream)
10: Update the local vector x using the new sample
11: **if** $x \notin \mathcal{B}$ **then** Report neighborhood violation and **return**
12: **if** $x \notin$ safe zone **then** Report safe zone violation and **return**
13: **procedure** NodeUpdateConstraint(safe zone, $\mathcal{B}$)
14: Update safe zone and the neighborhood $\mathcal{B}$

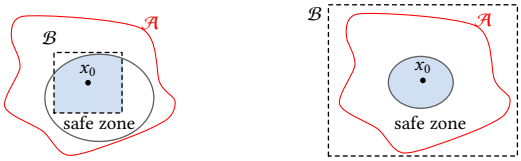
which is caused when the node's local vector is outside the safe zone. The second type is *neighborhood violation*, which is caused when the node's local vector is outside the neighborhood of x_0. The coordinator uses either type of ADCD to derive the ADCD local constraints. For ADCD-X, it applies Lemma 1 with $\mathcal{S} = \mathcal{B}$. In this case, the ADCD local constraints are based on eigenvalues that are evaluated inside a specific neighborhood. Hence, these local constraints are applicable only to this neighborhood: they maintain correctness as long as all the local vectors are inside $\mathcal{B}$. Nodes must therefore first determine that the local vector is inside $\mathcal{B}$ before checking for a safe zone violation. Note that when using ADCD-E, the coordinator uses Lemma 2; in this case, the neighborhood is the entire f's domain $\mathcal{D}$ since the Hessian is constant.

At initialization, the coordinator first determines whether to use ADCD-X or ADCD-E, depending on the function to approximate f (§3.1 and §3.2). If ADCD-E is used, the coordinator evaluates H^- and H^+. If ADCD-X is used, the coordinator uses the designed approximation error bound ϵ to evaluate the optimal neighborhood size $\hat{r}$ using the tuning algorithm presented in the next section.

The node algorithm is simple: whenever there is an update to the local vector, the node will check if the data remains in the neighborhood $\mathcal{B}$ and whether the ADCD local constraints still hold. If not, the node will report a violation to the coordinator. Otherwise, the node does nothing. Whenever an updated local constraint arrives from the coordinator, it will use the new constraint.

Lazy Sync and Slack: We incorporate two commonly-used enhancements to the above protocol, which help reduce the number of local violations and allow the coordinator to resolve them without pulling local vectors from all the nodes. As they are not the focus of this work, we only include necessary detail, and refer the reader to prior work [22, Sec. 4.2 and 4.6] for description and analysis.

With *slack*, nodes add a *slack vector* s^i to the local vector x^i when checking the local constraints (Alg. 1, lines 11 and 12); s^i is set by the coordinator to $x_0 - x^i$ whenever we update x_0 (line 2). *Lazy sync* is an incremental approach to resolving safe zone violations. When a violation is reported to the coordinator, it starts adding nodes one by one to a *balancing set* $\mathcal{S}$ until either the violation is resolved (in which case it rebalances s^i for the nodes in $\mathcal{S}$ and monitoring resumes without changing x_0), or until $|\mathcal{S}| > \frac{n}{2}$ (in which case it falls back to the full sync in line 1). We use a least-recently used (LRU) strategy to select nodes to add to $\mathcal{S}$. We explore the impact of slack and lazy sync in §4.6.



(a) Smaller neighborhood. **(b) Larger neighborhood.**

Figure 2: Tradeoff between neighborhood size (dashed rectangle) and the resulting safe zone size (solid circle). The local constraint is their intersection (shaded area).

3.6 Setting the Neighborhood Size

ADCD-X requires finding the extreme eigenvalues in a neighborhood $\mathcal{B}$ of size r around x_0. The choice of neighborhood size r is important since it affects the eventual efficiency of the ADCD local constraints. An increase in r leads to increase in the search domain for $\lambda_{\min}$ and $\lambda_{\max}$, which can results in more extreme eigenvalues than a smaller r produces, resulting in different DC decomposition.

Interestingly, while prior work observed that different constraints are optimal in different regions of the data space [41], the question of neighborhood size did not come up. Prior work focused on finding analytically-derived constraints designed to be *globally correct*: correct everywhere in f's domain $\mathcal{D}$. In contrast, ADCD provides a neighborhood around the reference point, which in turn means the ADCD local constraints need only be correct for data inside the neighborhood $\mathcal{B}$. Hence, ADCD constraints are *locally correct*.

This presents us with a new opportunity: since ADCD constraints need only be locally correct, they can be more permissive, resulting in fewer safe zone violations. The challenge lies in balancing the tradeoff between neighborhood and safe zone violations. If the neighborhood $\mathcal{B}$ is very small (small r), the resulting safe zone can be large, but local data easily moves outside the neighborhood, which means more *neighborhood violations (i.e., $x \notin \mathcal{B}$)*. If $\mathcal{B}$ is very large (large r) there will be few neighborhood violations, but the resulting safe zone might be needlessly restrictive due to more extreme eigenvalues, resulting in many safe zone violations. Figure 2 illustrates this tradeoff. In Figure 2(a) the neighborhood $\mathcal{B}$ is very small, resulting in a large safe zone but potentially many neighborhood violations. In Figure 2(b) $\mathcal{B}$ is very large and in fact $\mathcal{B}$ is a super-set of the admissible region $\mathcal{A}$ and hence there can be no neighborhood violations. However, this also results in a much smaller safe zone which could lead to many safe zone violations.

Effect of Neighborhood Size r: To explore the impact of the neighborhood size on the number of violations, we used AutoMon to monitor the Rozenbrock function $f(x) = (1-x_1)^2 + 100\left(x_2 - x_1^2\right)^2$, where x_1, x_2 are sampled from the normal distribution $\mathcal{N}(0, 0.2^2)$. We used additive approximation with approximation error bound ϵ: $L = f(x_0) - \epsilon, U = f(x_0) + \epsilon$. For a given approximation error bound ϵ we monitor the function with different values of r, and count the total number of neighborhood and safe-zone violations.

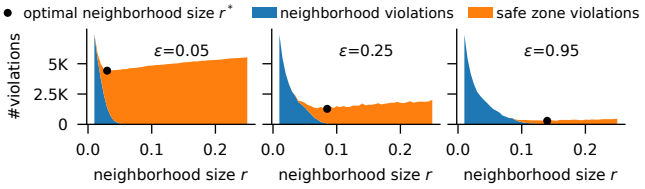


Figure 3: The effect of neighborhood size r on the number of violations while monitoring Rozenbrock function with four different approximation error bounds.

Algorithm 2 Neighborhood Size Tuning

$b \leftarrow 1$
while NoNeighborhoodViol(monitor with $r = b$) **do** $b \leftarrow b/2$
$lo \leftarrow b$, $hi \leftarrow b$
while AnySafezoneViol(monitor with $r = lo$) **do** $lo \leftarrow lo/2$
while AnyNeighborhoodViol(monitor with $r = hi$) **do** $hi \leftarrow 2 \cdot hi$
$R \leftarrow$ 10 equally spaced r values in the range $[lo, hi]$
return $\text{argmin}_{r' \in R}$ NumTotalViolations(monitor with $r = r'$)

Figure 3 shows the number of violations as a function of neighborhood size for four different approximation error bounds $\epsilon \in \{0.05, 0.25, 0.95\}$. For a specific ϵ, we observe the tradeoff between neighborhood violations and safe zone violations. Additionally, we see that permissive approximation error bounds (larger ϵ) imply larger safe zones, resulting in fewer safe zone violations. Increasing ϵ results in slightly more neighborhood violations, which we discuss below. Lastly, we observe that neighborhood violations decrease when the neighborhood size increases, as expected.

Safe zone and neighborhood violation can hide each other. Since the nodes check for neighborhood violation before checking the ADCD local constraint, some of the safe zone violations are concealed by neighborhood violations. However, the opposite also happens. When ϵ is small, the resulting small safe zone leads to many safe zone violations. When these are resolved, the coordinator updates x_0 and the neighborhood $\mathcal{B}$, meaning that a future neighborhood violations are less likely. Therefore smaller ϵ results in fewer neighborhood violations.

Tuning Procedure: The optimal neighborhood size r^*, shown in Figure 3 as a dot, is the neighborhood size that obtains the smallest number of violations in total. The optimal size r^* depends on the function, the data, and the allowed approximation error ϵ.

To avoid the user needing to specify the neighborhood size r, AutoMon automatically tunes for the approximated optimal size $\hat{r}$. This is done by running AutoMon on a small subset of the initial data and counting violations. Algorithm 2 is the tuning algorithm to find the approximated optimal neighborhood size $\hat{r}$. It finds a low neighborhood size r for which there are no safe zone violation and a high r with no neighborhood violations, and then returns a neighborhood size in between with fewest total violations. We evaluate the effectiveness of this tuning procedure in §4.5.

Since tuning is done on a small subset of the data, later changes in data distribution can mean $\hat{r}$ found by the tuning process becomes too small, causing unnecessary neighborhood violations. In our experience this is rare, mostly when the error bound is very large. We mitigate this using a simple heuristic: whenever the coordinator observes $5n$ consecutive neighborhood violations with no intervening safe zone violations, it multiplies $\hat{r}$ by 2. We leave adaptive tuning of r to future work.

3.7 Assumptions and Correctness Guarantees

AutoMon's correctness guarantees are given under three core assumptions. First, we make the mild assumption that automatic differentiating obtains accurate Hessians. Second, we assume nodes and coordinator communicate using an underlying messaging fabric which guarantees delivery. Third, we assume that the rate in which each node receives local data is lower than the maximum time to resolve violations, which depends on the network latency and the time it takes the coordinator to compute local constraints.

Under these assumptions, AutoMon provides a deterministic correctness guarantee if the representation used to derive the ADCD local constraints is a true DC decomposition in $\mathcal{B}$, i.e. $\check{g}, \check{h}$ are convex or $\hat{g}, \hat{h}$ are concave in $\mathcal{B}$. In this case, the local constraints are convex (§3.3). This convexity implies that if all local vectors x^i are within AutoMon's safe zone, then any convex combination of x^i, including $\bar{x} = \frac{1}{n}\sum x^i$, is inside the safe zone, thus $L \leq f(\bar{x}) \leq U$.

Therefore, ADCD provides strong correctness guarantee when approximating functions with constant Hessian, as ADCD-E obtains true DC decomposition. In addition, ADCD-X provides correctness guarantee when approximating convex and concave functions. For any convex function the minimal eigenvalue of $H(x)$ at every $x \in \mathcal{D}$ is non-negative. Hence $\hat{\lambda}$ found by the optimization process is non-negative and $\lambda^-_{\min} = 0$. Since $0 \leq \lambda^+_{\max}$, the DC Heuristic chooses the convex difference, which is a true DC decomposition as $\lambda^-_{\min}$ is a lower bound for $\lambda_{\min}$.

ADCD-X does not necessarily guarantee correctness for other arbitrary functions since the optimization problem (3) may converge on a local solution; inaccurate $\lambda^-_{\min}$ and $\lambda^+_{\max}$ values can result in representation that is not a DC decomposition. We mitigate this using a simple sanity check. Recall that by construction, AutoMon's safe zone defined by the local constraints is included in the admissible region. Thus, whenever the local vector x is inside the safe zone, nodes also verify that $L \leq f(x) \leq U$ (i.e., $x \in \mathcal{A}$). In the rare case where this verification fails, the node notifies the coordinator about a violation and indicates that the local constraints are faulty; the coordinator then initiates a full sync. Our evaluation shows AutoMon provides a good approximation for even highly non-convex functions with discontinuous derivatives such as neural networks with ReLU activations (§4).

3.8 Library API

AutoMon is not a complete distributed data processing system. Like sketches (§5), AutoMon is an algorithmic building block for building such systems (potentially using existing software frameworks [13]). We therefore design AutoMon as an agnostic library that focuses strictly on the monitoring algorithm. Application details and system-level support for reading data, messaging, deployment, and so on are outside the scope of this library, and are the responsibility of the user. In particular, the developer must mediate

between AutoMon and a messaging fabric of their choice: the developer uses the library API to produce or consume message contents, which the messaging fabric transfers over the network.[6]

Given a function presented as a numeric program in a high-level language, AutoMon provides the basic API required to perform distributed monitoring of the function. The user first initializes an AutoMon node, `node = AutoMonNode(f, epsilon)`, passing the function to monitor and the required approximation. Retrieving the current approximated value of the function is simply a matter of calling the `node.current_value()` method which returns $f(x_0)$. The user must notify AutoMon when the local vector x has changed by calling `node.update_data(x)`, and send any resulting message (e.g., safe zone violation) to the coordinator; AutoMon will provide and process the contents of such messages. Similarly, when a message has been received from the coordinator the user must call `node.message_received(msg)`, then send back any reply.

4 EVALUATION

We empirically explore AutoMon's performance on several functions of increasing complexity, using both real-world and synthetic data. In particular, we investigate:

(1) The tradeoff between communication and approximation error.
(2) Scalability in vector length, runtime, and the number of nodes.
(3) The effectiveness of our neighborhood-size tuning procedure.
(4) The impact of AutoMon's features on the communication cost and accuracy: ADCD local constraints, slack, and lazy sync.
(5) Communication and bandwidth on a real-world WAN.

Our main performance metrics are communication and error. For communication, we focus on the number of messages sent during an experiment.[7] Since AutoMon is designed to keep $f(\bar{x})$ between L and U, we also measure the maximum error $|f(x_0) - f(\bar{x})|$.

We implemented the prototype in Python. It uses JAX [11] 0.2.1 for automatic differentiation (AD) to compute the Hessian and an L-BFGS-B solver from SciPy [63] 1.6.2 to find the extreme eigenvalues.

4.1 Experimental Setup

The input for each experiment is the source code for a function f and a dataset comprised of n data streams, one for each node in the system. The nodes maintain a sliding window over the data stream, and the local vector is defined as the average of the last W samples in the window. We emphasize that nothing in AutoMon requires a sliding window; it is concerned only with the local vector x.

We use discrete event simulation to simulate the distributed network on a single machine. Most experiments simulate a network of fixed-rate sensors: in each simulation round, every node reads a data update from its local stream, updates its local vector, and runs the AutoMon node code. For DNN intrusion detection experiments, only one node reads new data in each round of simulation, using the timestamps encoded in the dataset. During the experiment, we collect statistics about the messages that are sent between the nodes and the coordinator, the approximation error, missed violations, and so on. In all the experiments, we use AutoMon with Algorithm 2 for neighborhood-size tuning, lazy sync with LRU, and slack. Unless stated otherwise, we use 10 nodes for the synthetic datasets.

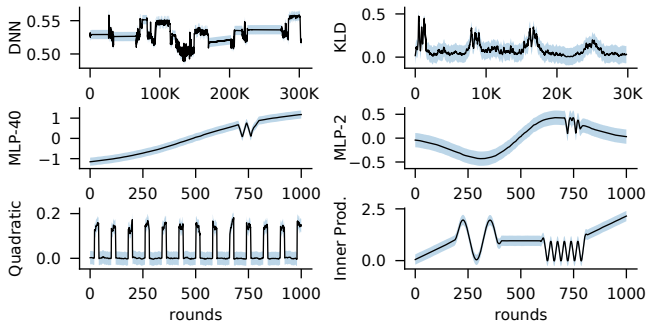


Figure 4: The functions and datasets used in evaluation. Lines show function value for the default dimension over time; shaded area shows approximation bounds $\pm\epsilon$.

We compare AutoMon with several baseline approaches:

- **Centralization:** Every node sends its local vector after every update. For many distributed functional monitoring tasks, including those that can use sketches, centralization is state-of-the-art since local changes require sending updated sketches [3].
- **Periodic:** Every node sends its local vector once every P simulation rounds (or P time units), where P is a period parameter. This approach is easy to reason about, but is not adaptive. It therefore suffers from many missed violations when the period is out of sync with the changes in the data.
- **CB:** Convex Bound [41] is a state-of-the-art GM-based approach. For functions with a CB local constraint [41], we include CB in our experiments. We run it with lazy sync and slack.

4.2 Functions and Datasets

We monitor different functions and use different datasets: synthetic for exploring the impact of different parameters on AutoMon and real-world, to evaluate AutoMon's performance over real data. For synthetic datasets, we used 200 rounds for neighborhood-size tuning data and run the experiment for 1000 rounds. For real data, the number of rounds is determined by the size of the dataset and we used $\sim 1.5\%$ of the data for tuning. We now describe the functions that we use and the dataset for each function. Figure 4 shows the value of each function during a run, as well as the additive error bound. For synthetic datasets, we use the default dimensions, number of nodes, and error bound.

- **MLP-d:** a neural network with an input layer of dimension d, three fully-connected hidden layers with tanh activation function, and a single output neuron with no special activation (identity function). The function is therefore:

$$f(x) = W_4 \cdot \varphi\left(W_3 \cdot \varphi\left(W_2 \cdot \varphi\left(W_1 \cdot x + b_1\right) + b_2\right) + b_3\right) + b_4\ ,$$

[6] We provide an example of a ZeroMQ [6] mediation layer in AutoMon's code repo.
[7] Though we include bandwidth measurements in §4.7, we focus on message count for several reasons. First, we are comparing algorithms, not systems. The number of messages is a common metric for comparing distributed algorithms because it is independent of any particular network setup and the underlying messaging stack, and because different functions and datasets have different dimensions [3]. The size of our payload is fixed and small; given the details above, inferring bandwidth is straightforward. Second, AutoMon is fully compatible with sketching techniques for reducing message size (see §5). Moreover, there is existing work on reducing the bandwidth of GM-based methods [3, 55]; we leave their implementation for future work. Lastly, in some settings, the number of messages can be more important than the bandwidth due to the power consumed by turning on the wireless radio [5, 61].

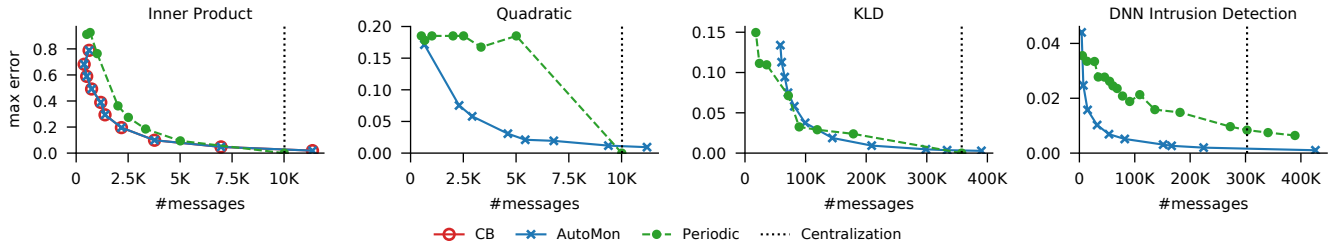


Figure 5: Error-communication tradeoff when monitoring different functions with several algorithms; lower and to the left is better. Each point represents the total number of messages (X axis) and maximum error (Y axis) in one monitoring run, with a specific setting of approximation error bound (or period, for Periodic).

where W_i and b_i are the respective weights and biases of the hidden layers, and φ is the tanh activation function, applied element-wise. We trained these weights to evaluate the function $x_1 \exp\left(-\frac{1}{d-1}\sum_{i=1}^{d} x_i^2\right)$, and monitor the output of the trained network $f(\bar{x})$. We use a synthetic dataset with x_1 sampled from the normal distribution $\mathcal{N}(\mu, 0.1^2)$ with the mean μ starting at -2 and increasing gradually over time. $x_2, ..., x_d$ are sampled from $\mathcal{N}(2, 0.1^2)$ for half the nodes, and we use $\mathcal{N}(-2, 0.1^2)$ for the remaining nodes. The data contains some outlier samples: the mean value of x_1 changes to 0 for 20 rounds starting from round 720, and then again from round 760. The default dimension d in our experiments is 40.

- **DNN for Intrusion Detection:** a deep neural network for intrusion detection, based on the KDDCup-99 dataset [1]. Each sample in the dataset represents a single network connection. The samples consist of 41 features and are labeled as either normal or an attack. For this classification task, we used a DNN with 5 fully-connected hidden layers comprising 512, 64, 32, 16, and 8 neurons in each layer, respectively, all using the ReLU activation function. The output layer contains a single neuron with sigmoid activation. After using the "10% KDD" dataset to train the network, the trained network achieved 0.933 accuracy, 0.98 precision, and 0.93 recall on the "Corrected KDD" test set [36]. We use the "Corrected KDD" test set as the data stream, resulting in a stream of 311029 samples. We divided the data into 9 local streams according to the application-type feature of the data. For applications that had many samples, we used a round-robin approach for load balancing and divided the samples between multiple nodes: "ECR_i" was divided between 5 nodes and "Private" was divided between 2 nodes. Another single node was responsible for all "Http" samples, and the last node was responsible for the other 62 applications.
- **KLD:** The Kullback–Leibler divergence function for discrete probability distributions P and Q, defined on the same probability space Ω: $D_{KL}(P\|Q) = \sum_{\omega\in\Omega} P(\omega)\log\left(P(\omega)/Q(\omega)\right)$. We use a real-world air pollutant dataset [72] collected hourly from $n = 12$ air-quality monitoring sites in Beijing over 4 years (34,536 data records per site). For each node (i.e., site), we used the PM10 attribute as P and the PM2.5 attribute as Q. Both attribute values are between 0 and 500, and we divided this range into $d/2$ bins, resulting in a local vector $x = [p, q]$, where $p, q \in \mathbb{R}^{d/2}$ are the local probability vectors for PM10 and PM2.5, respectively; we use a sliding window of size $W = 200$; in each round we update p and q with the new measurements. Since KLD is undefined when Q contains zero entries, we use a common variant which adds a small constant value τ to the entries of p and q before computing the function $f(x) = f([p, q]) = \sum_{i=1}^{d/2} p_i \log(p_i/q_i)$. We use $\tau = 1/(nW)$, the minimal possible value of the probability vectors in this setting. Since f is convex, AutoMon's approximation error is guaranteed. We control the dimension of the function by changing the number of bins $d/2$. By default, $d = 20$.
- **Inner Product:** $f(x) = f([u, v]) = \langle u, v\rangle$, with a synthetic dataset that contains quiet phases as well as rapid changes in the data. We generated the vectors $u, v \in \mathbb{R}^{d/2}$ such that $f([u, v])$ is a combination of a monotonic increasing function, a sine wave with low and high frequency, and a monotonic constant function. We control the dimension of the function by changing the dimension of u and v. By default, $d = 40$.
- **Quadratic Form:** $f(x) = x^T Q x$, where $Q \in \mathbb{R}^{dxd}$ is a random matrix with entries drawn from a standard normal distribution. AutoMon uses ADCD-E for this function since its Hessian is constant. We use synthetic data with each entry of $x \in \mathbb{R}^d$ sampled from the normal distribution $\mathcal{N}(0, 0.1^2)$. A single "outlier" node gets an alternating pattern: 40 samples drawn from $\mathcal{N}(0, 0.1^2)$ followed by 40 samples drawn from $\mathcal{N}(-10, 0.1^2)$. We use $d = 40$.

For KLD, the node receives the frequency vector of the samples in the sliding window and the sliding window size is 200 samples. For the other functions, the node receives the average vector of the samples in the sliding window and the sliding window size is 20 samples. We start updating the nodes with data only after all the sliding windows of all the nodes are full.

4.3 Error-Communication Tradeoff

There is an inherent tradeoff between the approximation error and the resulting communication. A communication-efficient algorithm should use minimal communication and produce a low error.

We compare AutoMon with the baselines for four functions, two with synthetic and two with real-world datasets. In each run, we monitor the functions using a different approximation error bounds (or period values for Periodic), count the total number of messages and the maximum approximation error, and plot the resulting tradeoff curve. The best algorithm is the one that achieves

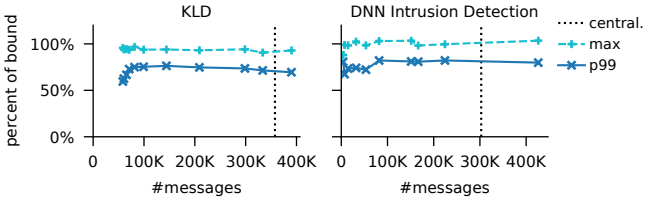
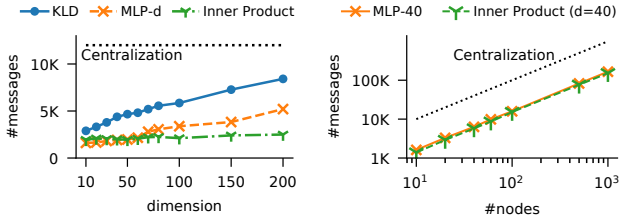


Figure 6: AutoMon communication and error for KLD and DNN. Max (dashed) and 99th percentile (solid) error are shown as percentage of the requested approximation bound.



(a) Impact of dimension d. **(b) Impact of number of nodes n.**

Figure 7: Impact of dimension and the number of nodes on AutoMon's communication.

the lowest error and communication. Figure 5 shows the tradeoff curves of AutoMon and the different baselines.

AutoMon exhibits the best tradeoff overall. It produces a low error at a fraction of the number of messages needed by Centralization. It also uses fewer or a similar amount of messages to those required by the non-adaptive Periodic algorithm.

Inner Product: AutoMon automatically achieves identical performance to the carefully tailored CB-based approach of Lazerson et al. [41]. The CB-based algorithm uses the following identity to represent the inner product as a difference between two convex functions: $\langle x, y\rangle = \frac{1}{4}\|x+y\|^2 - \frac{1}{4}\|x-y\|^2$. This form is equivalent to ADCD-E (due to space limitation, the proof is omitted). While [41] provides a CD representation for a specific functions, AutoMon automatically finds such a representation for any function.

Quadratic Form: This function's value can change rapidly, as shown in Figure 4. The only way to obtain a small approximation error with Periodic is to have a period of 1 (i.e., Centralization); any larger period will result in high approximation errors. In contrast, AutoMon is adaptive and can provide a smooth, superior, tradeoff between communication and efficiency.

KLD: AutoMon yields a tradeoff that is similar to the Periodic algorithm, but with a smoother, better-controlled tradeoff curve. Recall however, that the Periodic algorithm is non-adaptive, and the curves in Figure 5 are derived post-hoc. Conversely, AutoMon is adaptive and can handle changes in the data distribution. Moreover, as described above, AutoMon provides a deterministic guarantee for KLD, unlike the Periodic algorithm.

DNN: AutoMon sends fewer messages than Periodic across all ϵ values. Unlike the other datasets where local data for all nodes updates every round, the DNN dataset only updates a single node at a time. This means the function changes gradually, which AutoMon exploits, while the non-adaptive Periodic sends updates from all nodes once per period even when changes are small. Moreover, since the period parameter now represents a time interval (number of rounds) rather than the number of data samples, Centralization uses fewer messages than Periodic with a period of 1.

Error Relative to Bound: Figure 6 shows AutoMon's relative error with respect to the error bound ϵ for KLD, where AutoMon guarantees the approximation accuracy (§3.7), and for DNN, where there is no such guarantee. Despite the lack of guarantee, DNN's error profile is similar to KLD: In practice AutoMon's error is below the approximation bound 99% of the time for both functions. Even in the rare cases for DNN when the maximum error is above the bound, it is still very close.

4.4 Scalability: Dimensions, Nodes, Runtime

We use the KLD, Inner Product, and MLP-d functions to study how data dimensionality affects the communication and runtime of AutoMon, as well as its ability to scale with the number of nodes. For every function, we carry out multiple runs with input dimension $d \in [10, 200]$. As shown below, at high dimensions the bottleneck becomes the numerical optimization in full sync. To compare the different functions and datasets, we set the number of nodes to $n = 12$ and stop each run exactly 1000 rounds after the first sliding window is full, resulting in a centralized cost of 1000 messages per node. Runtimes are measured on an Intel i9-7900X at 3.3GHz with 64GB RAM, running Ubuntu 18.04 with MKL 2019 Update 3.

Dimensionality: Prior work has shown that the size of the local vector x can increase the number of messages needed to monitor a function [21]. However, for AutoMon, the function itself is also a parameter, and we explore how their combination impacts communication. Figure 7(a) shows the total number of messages in each run for different functions and input dimensions. We observe that AutoMon's scalability is highly dependent on which function is being monitored. While communication increases with dimension for all functions, this increase is minimal for Inner Product, moderate for MLP-d, and more drastic for KLD. Nevertheless, even for KLD, we observe that AutoMon remains better than centralization for up to 200 dimensions.

Number of Nodes: More nodes means more communication, but this growth is contingent upon the distribution of the data between different nodes. Figure 7(b) illustrates how the number of messages grows with the number of AutoMon nodes for Inner Product ($d = 40$) and MLP-40. While the number of messages does increase with the number of nodes in the system, we observe that the same happens for Centralization, and that the ratio between Centralization and AutoMon is fixed. In these synthetic datasets, the data of new nodes is similar to the data of existing nodes. Therefore, the probability for violation of a single node does not change with the number of nodes, neither does the probability of resolving violations using lazy sync, which explains the fixed ratio. We therefore conclude that the AutoMon technique does not limit the scalability of the system. This limitation may emerge from the data itself.

Node Runtime: AutoMon's node runtime should be low since it is targeting environments where the computational power of local data sources is low. We measured the time a node takes to check a single data update, as well as the time it takes the node to complete different tasks during the data update process (figures

omitted for lack of space). The impact of the dimension on the average runtime is negligible, on average 1 millisecond or less for all functions and dimensionality. The time to verify that the local vector is inside the safe zone is close to the time it takes to simply evaluate the original function on the local vector, ranging from 0.01 to 1 millisecond. We therefore conclude that AutoMon node is suitable even for computationally limited edge devices.

Coordinator Runtime: While AutoMon's coordinator may not be a resource-constrained edge device, the coordinator's runtime limits the data rate supported by AutoMon because nodes must wait for the coordinator to resolve violations. This runtime is dominated by the full sync with ADCD-X, which requires solving a numerical optimization problem to find the extreme eigenvalues. The lazy sync time is orders of magnitude smaller, as it only requires evaluating the local constraints. For KLD and MLP-d, which use ADCD-X, the average time for the full sync increases with the dimension, ranging from 0.2 seconds ($d = 10$) to 12 seconds ($d = 200$). For Inner Product, the coordinator uses ADCD-E, where eigendecomposition is done only once at initialization; full sync time is below 10 milliseconds for all dimensions. (Figure omitted for lack of space.)

4.5 Impact of Neighborhood Size Tuning

To demonstrate the effectiveness of Algorithm 2, we show that (1) the $\hat{r}$ found by the algorithm is close to the true optimal neighborhood size r^*; (2) r can have a large impact on AutoMon's communication; (3) no single fixed r is optimal across different approximation error bounds ϵ; and (4) the tuning procedure yields comparable performance to using the optimal r^*.

To evaluate Algorithm 2 we used AutoMon with a range of approximation error bounds ϵ and neighborhood sizes r to monitor the MLP-2 function, as well as the Rozenbrock function, defined as $f(x) = (1-x_1)^2 + 100\left(x_2 - x_1^2\right)^2$. We chose this function because it is especially challenging for gradient-based numerical approaches (e.g., AutoMon or gradient descent), and because its Hessian is non-constant. To generate data, we draw entries x_1, x_2 from the normal distribution $\mathcal{N}(0, 0.2^2)$. For each approximation error bound ϵ, we find the optimal neighborhood size r^* that obtains the smallest number of violations, as well as the recommended neighborhood size $\hat{r}$ found by the tuning procedure. We repeated each experiment 5 times, sampling a new dataset every run.

On average, the true optimal neighborhood size r^* is close to the neighborhood size found by the tuning procedure $\hat{r}$ (figure omitted due to lack of space), especially given the significant effect of the randomness in the data. The mean relative error of the tuning algorithm with respect to the optimal value is 8% for Rozenbrock and 20% for MLP-2. On average, we find that $\hat{r}$ found by Algorithm 2 is within 1.03 standard deviations of the optimal r^* for both functions. Rozenbrock is highly sensitive to small input changes by design. Hence, the standard deviation for $\hat{r}$ is large while r^* has a small range (as Figure 8 shows). Conversely, MLP-2 is less sensitive. It has a wider range of optimal r^*, while the tuning procedure tends to converge to same $\hat{r}$.

To evaluate the impact of r on communication, we run AutoMon over the sampled datasets, each with its optimal r^* and the tuned $\hat{r}$, as well as three fixed neighborhood sizes $r \in \{0.05, 0.5, 2.5\}$. Figure 8 shows AutoMon's average communication for each approximation

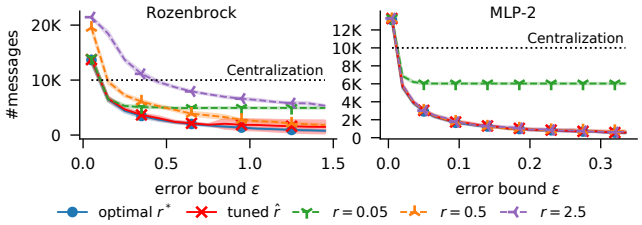


Figure 8: Mean number of messages for different approximation error bounds ϵ, while using optimal neighborhood size r^*, tuned size $\hat{r}$, and fixed r during monitoring. The standard deviation is small and barely visible (shaded area).

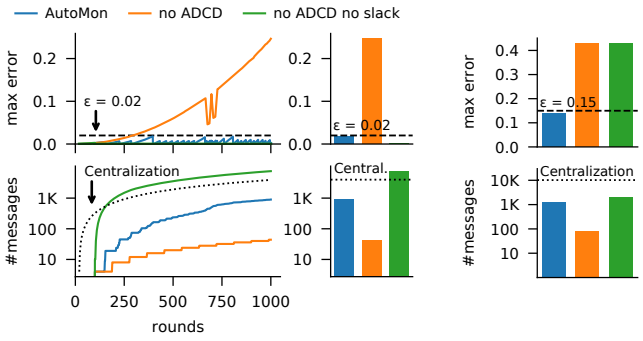
error bound ϵ. We make three observations. First, using the wrong r can substantially increase communication. Second, no single r is best across all ϵ and functions. For example, for Rozenbrock the average increase in the number of messages when using $\hat{r}$ instead of r^* is 33%, while for the fixed r it grows by more than 100%. For MLP-2, the difference between r^* and $\hat{r}$ is 3.5%; however, for the best other fixed r it is more than 7%. The results for MLP-2 also suggest there is a range of neighborhood sizes that work well; however, this range is not known a-priori. Third, and most crucial, we observe that using the $\hat{r}$ found by the tuning process results in a similar number of messages as using the optimal neighborhood r^*.

4.6 Impact of ADCD, Slack, and Lazy Sync

We perform an ablation study to evaluate the contribution of different components of AutoMon. Could we replace the ADCD local constraints with simply checking the global condition on local data $L \le f(x) \le U$, relying on the GM protocol to reduce communication? As shown below, even for simple functions with only few nodes, this is not the case.

We demonstrate this using the function $f(x) = -x_1^2 + x_2^2$ with four nodes. We simulate 1000 rounds with the local data for the four nodes initially the same, starting at $(x_1, x_2) = (0, 0)$. As the experiment progresses, local node data slowly drifts in different directions, which is common in distributed setting. Specifically, the local vectors move towards $(1, 0)$, $(-1, 0)$, $(1, 1)$ and $(1, -1)$). For two nodes, we also add outliers between rounds 650 and 700.

Figure 9(a) shows the approximation error (top) and cumulative messages (bottom) of each algorithm over time. AutoMon maintains the desired approximation error using minimal communication, since the ADCD local constraints for f are both correct and efficient. Without ADCD, however, the monitoring suffers from missed violations: locally for every node i, $L \le f(x^i) \le U$, yet globally $L > f(\bar{x})$ or $f(\bar{x}) > U$. This happens because lazy sync manages to balance the slack of different nodes, preventing the global sync from recomputing the reference point x_0. The end result is unbounded and ever-increasing error, albeit with little communication. We further removed slack and lazy sync, which gave us a basic GM protocol: similar to Algorithm 1 but without ADCD. This results in a low approximation error due to many full sync operations, but at the cost of more communication than would be used by a centralization approach (which would result in no error).



(a) $f(x) = -x_1^2 + x_2^2$. (b) MLP-2.

Figure 9: Impact of AutoMon features on approximation error (top) and cumulative communication (bottom) when monitoring $f(x) = -x_1^2 + x_2^2$ (left) and MLP-2 (right); lower is better. ADCD and slack are both needed to achieve low error and low communication.

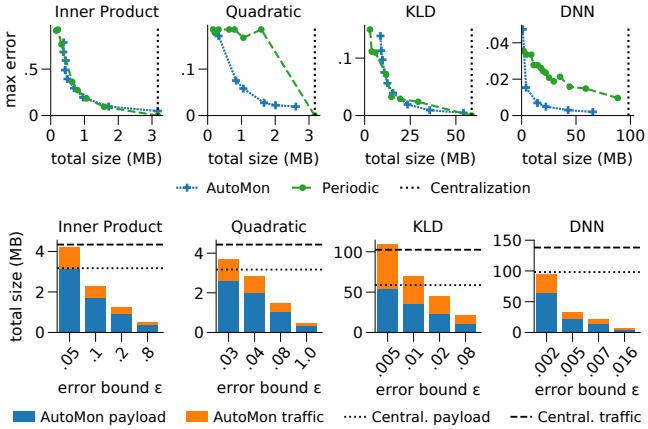


Figure 10: Bandwidth usage in distributed experiments over public WAN. Top: error-bandwidth tradeoff, with X axis showing total size of message payload for each monitoring approach. Bottom: AutoMon's total message payload size and measured network traffic for a range of ϵ values from the distributed experiments in Figure 10. Horizontal lines depict centralization payload size and network traffic.

We repeated the experiment with the more complex MLP-2 function and present the results in Figure 9(b). Without ADCD, the approximation error grows to over twice the size of the bound. Unlike before, removing the slack and lazy sync mechanism does not help since for MLP-2 it is even more likely that locally $L \leq f(x^i) \leq U$ while globally $L > f(\bar{x})$ or $f(\bar{x}) > U$. In contrast, AutoMon maintains the desired approximation with low communication by synchronizing as needed.

4.7 Validation on Real-World Deployment

Simulations let us compare algorithms while controlling important variables (such as n, d) in isolation from confounders (e.g., choice of network stack). We now verify our simulation in Section 4.3 through a series of geo-distributed experiments. We conducted each run in our experiments on two Amazon ECS clusters [4]: one is located in us-west-2 region and is comprised of a single coordinator using 16 vCPUs and 32GB of memory on an Intel Xeon CPU at 3.4–3.9 GHz; the other is located in us-east-2 and includes all nodes, each with 1 vCPU and 4GB of memory on an Intel Xeon CPU at 2.2–2.5GHz; the average RTT was 56ms. For messaging, we used ZeroMQ [6]. We set the time between data updates to 5 seconds for DNN and to 1 second for the other datasets. We count the total bytes in the payload of AutoMon's messages, and use Nethogs [17] to monitor the traffic volume of the coordinator process, which includes both payload and overheads such as ZeroMQ and packet headers.

Number of Messages: Real-world communication matches our simulation, with a median difference of 0% for the DNN function, less than 5.3% for inner product and KLD, and 16.6% for Quadratic (figure omitted). Slight timing differences when nodes update their local data result in a small number of additional messages, as the coordinator requests the local vectors for nodes after they had already reported a local violation.

Error-Bandwidth Tradeoff: The top of Figure 10 shows the error as a function of total payload size. The error-bandwidth tradeoff generally agrees with the error-communication tradeoff in Figure 5. The relative ranking of methods is also the same: AutoMon's total payload size is lower than that of Periodic whenever AutoMon uses less messages than Periodic.

Network Traffic Consumed: The bottom of Figure 10 shows AutoMon's total payload size (blue) and actual network usage (orange) for a range of ϵ values. The dotted line is Centralization's payload size and the dashed line is its network traffic. In most cases, except in the extreme cases for a very small ϵ value, AutoMon's usage is less than Centralization's payload. It reduces data transfer volumes by up to 98%, depending on the requested error bound ϵ, and by 65% on average across all tested functions and error bounds.

5 RELATED WORK

We divide related work on distributed monitoring into five areas.

Sketches: Sketches are comprised of a sketching procedure to reduce the data size, and an appropriate query function that estimates a statistic using the sketch [54]. They substantially reduce the size of messages required to monitor a function, while offering (usually probabilistic) approximation guarantees. Unlike AutoMon, sketches are generally tailored for specific functions and queries; (e.g., PCA [31]); creating a sketch for a new function is non-trivial, requiring manual effort and significant mathematical sophistication [45, 46, 70]. Notably, AutoMon is compatible with most sketches in the turnstile model, since they are linear or can be made linear [43]. AutoMon can monitor a linear sketch by defining f as the query function and x as the sketched data structure, since $\bar{x} = 1/n \sum x^i$.

Generic Sketches: Universal Sketches [12] provide a distributed approximation for any function from the *Stream-PolyLog* family using a single universal sketch data structure, while requiring no more sophistication than being able to compute the desired function. Specifically, if x is a vector of counts (frequencies), and $f(x) = \sum g(x_i)$ where x_i are frequency counts and g is monotonic and

bounded from above by $O(x_i^2)$, then given an implementation of g universal sketches provide a multiplicative approximation for $f(x)$ with probabilistic guarantees.

Universal sketches are heavily used in the UnivMon framework for network flow monitoring [45]. Similarly, the AutoMon library is an application-agnostic building block for distributed applications and frameworks. Though similar in spirit, universal sketches and AutoMon have different constraints, guarantees, and performance metrics. First, they are limited to *Stream-PolyLog* functions defined over frequency vector in the turnstyle stream model. Conversely, AutoMon supports a much wider class of functions and the data vector x can be defined arbitrarily. Second, universal sketches focuses on providing strong probabilistic guarantees on accuracy. While AutoMon does provide strong deterministic accuracy guarantee for functions with constant Hessian and for convex and concave functions, we also show empirically that it is accurate even when no such guarantee is provided. Finally, sketches can reduce the size of each message (by reducing the size of the sketch), while AutoMon focuses on reducing the number of messages exchanged.

Nitrosketch [44] is a general framework for accelerating the computation time of existing sketches such as universal sketches; it does not address designing those sketches automatically.

General Algorithms for Distributed Monitoring: While many works propose distributed algorithms for monitoring specific functions, they tend to use bespoke protocols; applying such methods (e.g., distributed counting) to new functions (e.g., entropy) often requires non-trivial effort and development of new techniques [14].

Some works focus on providing general approaches for distributed function monitoring. Geometric Monitoring (GM) [40, 57] is a family of communication-efficient approaches to distributed monitoring that share the same underlying protocol of using convex local constraints to monitor a global threshold condition. These have been used to approximate diverse functions including variance [23], mutual information [27], AMS sketches [25], linear regression [21], and more [20, 28, 38, 42, 56]. Convex Bound [41] leverages ideas from GM as well as *DC decompositions* [2] to monitor several non-convex functions; however, again this approach requires mathematical sophistication and cannot be applied automatically. Samoladas and Garofalakis [55] introduce Functional Geometric Monitoring, which replaces the GM protocol with a distributed counting protocol, greatly reducing the size of messages. As with other methods, it requires finding local constraints (*safe functions*) for each new monitored function. More recently, Alfassi et al. [3] proposed a "drop-in" replacement of the GM protocol to reduce its bandwidth, while relying on the existing local constraints.

Though general, none of these are automatic; they require in-depth mathematical analysis to develop local constraints for new functions. Gabel et al. [22] show how to apply GM automatically but their approach is limited to convex or concave functions. Conversely, AutoMon derives its local constraints automatically for arbitrary functions of the global vector, directly from source code.

Distributed Dataflow and Query Planning: Stream processing engines [13, 71] execute distributed computation as a data-flow graph of built-in primitive operators. Other approaches optimize the aggregation network that runs a given query over distributed data [29, 48, 66]. Such techniques require expressing the computation using only a limited set of built-in primitives [69]. For complex numerical functions such as f_{nn} in §1 or the DNN in §4 they are equivalent to centralization or periodic updates. AutoMon can complement these approaches by optimizing user-defined operators.

Geo-Distributed Data Analytics: Systems proposed for analyzing geo-distributed data generally fall into one of the above general approaches [39, 48, 62, 64], or are designed for specific tasks using bespoke techniques that do not readily generalize [30, 34, 52].

6 DISCUSSION

AutoMon is an easy-to-use algorithmic building block for automatically approximating arbitrary real multivariate functions over distributed data streams. Given a source code snippet of an arbitrary function of the global aggregate, AutoMon automatically provides communication-efficient distributed monitoring of the function approximation, without requiring any manual analysis by the user. Our evaluation on synthetic and real-world datasets shows that AutoMon's error-communication tradeoff is comparable to previous hand-crafted algorithms, while using up to 50 times fewer messages on functions for which such efficient algorithms are not known.

Future work will concentrate on addressing AutoMon's limitations, and on improving its accuracy and performance.

First, AutoMon requires that f be a function of the average vector $\bar{x}$, and does not capture functions such as $\sum_i \sum_j x^i x^j$ used in support vector machines [60]. Though many functions can be rewritten in terms of $\bar{x}$ [21, 22, 25, 40, 41, 51], this is currently done manually. We plan to explore automatic function rewriting, as well as support for more aggregations (e.g., max, sum of inner products).

Second, for functions not covered by the guarantees in §3.7 the numerical optimization could yield inaccurate extreme eigenvalues and, therefore, violation of the error bounds. Bounding the Hessian eigenvalues [47, 50] can alleviate this issue. We also intend to study what factors impact AutoMon's performance. The error-communication tradeoff is determined by both the function as well as the data and window size. For example, when the extreme eigenvalues are exceptionally large/small, the derived safe zone can be very small leading to many safe zone violations. Inferring this *a priori* is hard for complex, hard-to-analyze functions – if we could easily understand their behavior analytically, we would not need AutoMon in the first place. However, we can use such observations to improve performance by switching on the fly to other monitoring approaches (e.g. Periodic).

Lastly, approximation error can be high if numerical optimization in the coordinator takes too long, which limits incoming data rate (§3.7, §4.4). To scale AutoMon to higher dimensions and data rates, we plan to explore Hessian spectrum approximations [26], as well as pre-computing future constraints when the coordinator is idle.

ACKNOWLEDGMENTS

The authors thank Ran Ben Basat and the anonymous reviewers for their valuable feedback. The research leading to these results was supported by the Israel Science Foundation (grant No.191/18). This research was partially supported by the Technion Hiroshi Fujiwara Cyber Security Research Center, the Israel National Cyber Directorate, and the HPI-Technion Research School.

REFERENCES

[1] 1999. Knowledge discovery in databases DARPA archive. Task Description. http://kdd.ics.uci.edu/databases/kddcup99/task.html.

[2] Amir Ali Ahmadi and Georgina Hall. 2018. DC Decomposition of Nonconvex Polynomials with Algebraic Techniques. *Math. Program.* 169, 1 (May 2018), 69–94.

[3] Yuval Alfassi, Moshe Gabel, Gal Yehuda, and Daniel Keren. 2021. A Distance-Based Scheme for Reducing Bandwidth in Distributed Geometric Monitoring. *37th IEEE International Conference on Data Engineering* (2021).

[4] Amazon. 2021. Amazon Elastic Container Service. https://aws.amazon.com/ecs

[5] Giuseppe Anastasi, Marco Conti, Mario Di Francesco, and Andrea Passarella. 2009. Energy conservation in wireless sensor networks: A survey. *Ad Hoc Networks* 7, 3 (2009), 537 – 568.

[6] The ZeroMQ authors. 2021. ZeroMQ. https://zeromq.org

[7] Atılım Günes Baydin, Barak A. Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. 2017. Automatic Differentiation in Machine Learning: A Survey. *J. Mach. Learn. Res.* 18, 1 (Jan. 2017), 5595–5637.

[8] Ran Ben Basat, Gil Einziger, Isaac Keslassy, Ariel Orda, Shay Vargaftik, and Erez Waisbard. 2018. Memento: Making Sliding Windows Efficient for Heavy Hitters. In *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies (CoNEXT '18)*. 254–266.

[9] Kanishka Bhaduri and Hillol Kargupta. 2008. *An efficient local Algorithm for Distributed Multivariate Regression in Peer-to-Peer Networks.* 153–164.

[10] Stephen Boyd and Lieven Vandenberghe. 2004. *Convex Optimization.*

[11] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. 2018. *JAX: composable transformations of Python+NumPy programs.* http://github.com/google/jax

[12] Vladimir Braverman and Rafail Ostrovsky. 2010. Zero-One Frequency Laws. In *Proceedings of the Forty-Second ACM Symposium on Theory of Computing (STOC '10)*. 281–290.

[13] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache Flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 36, 4 (2015).

[14] Graham Cormode. 2013. The Continuous Distributed Monitoring Model. *SIGMOD Rec.* 42, 1 (May 2013), 5–14.

[15] Graham Cormode, S. Muthukrishnan, and Ke Yi. 2011. Algorithms for Distributed Functional Monitoring. *ACM Trans. Algorithms* 7, 2, Article 21 (March 2011), 20 pages.

[16] Chuck Cranor, Theodore Johnson, Oliver Spataschek, and Vladislav Shkapenyuk. 2003. Gigascope: A Stream Database for Network Applications. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data* (San Diego, California) *(SIGMOD '03)*. Association for Computing Machinery, New York, NY, USA, 647–651.

[17] Arnout Engelen. 2020. Nethogs. https://github.com/raboof/nethogs

[18] Seyed K. Fayaz, Yoshiaki Tobioka, Vyas Sekar, and Michael Bailey. 2015. Bohatei: Flexible and Elastic DDoS Defense. In *Proceedings of the 24th USENIX Conference on Security Symposium (SEC'15)*. 817–832.

[19] Flink. 2015. User-Defined Functions. https://ci.apache.org/projects/flink/flink-docs-release-1.13/docs/dev/datastream/user_defined_functions

[20] Arik Friedman, Izchak Sharfman, Daniel Keren, and Assaf Schuster. 2014. Privacy-Preserving Distributed Stream Monitoring. In *21st Annual Network and Distributed System Security Symposium NDSS*.

[21] Moshe Gabel, Daniel Keren, and Assaf Schuster. 2015. Monitoring Least Squares Models of Distributed Streams. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '15)*. 319–328.

[22] Moshe Gabel, Daniel Keren, and Assaf Schuster. 2017. Anarchists, Unite: Practical Entropy Approximation for Distributed Streams. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '17)*. 837–846.

[23] M. Gabel, A. Schuster, and D. Keren. 2014. Communication-Efficient Distributed Variance Monitoring and Outlier Detection for Multivariate Time Series. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. 37–47.

[24] S. García, M. Grill, J. Stiborek, and A. Zunino. 2014. An Empirical Comparison of Botnet Detection Methods. *Comput. Secur.* 45 (Sept. 2014), 100–123.

[25] Minos Garofalakis, Daniel Keren, and Vasilis Samoladas. 2013. Sketch-Based Geometric Monitoring of Distributed Stream Queries. *Proc. VLDB Endow.* 6, 10 (Aug. 2013), 937–948.

[26] Behrooz Ghorbani, Shankar Krishnan, and Ying Xiao. 2019. An Investigation into Neural Net Optimization via Hessian Eigenvalue Density. In *Proceedings of the 36th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 97)*, Kamalika Chaudhuri and Ruslan Salakhutdinov (Eds.). 2232–2241.

[27] Nikos Giatrakos, Antonios Deligiannakis, Minos Garofalakis, Izchak Sharfman, and Assaf Schuster. 2012. Prediction-Based Geometric Monitoring over Distributed Data Streams. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data (SIGMOD '12)*. 265–276.

[28] Nikos Giatrakos, Antonios Deligiannakis, Minos Garofalakis, Izchak Sharfman, and Assaf Schuster. 2014. Distributed Geometric Query Monitoring Using Prediction Models. *ACM Trans. Database Syst.* 39, 2, Article 16 (may 2014), 42 pages.

[29] Arpit Gupta, Rob Harrison, Marco Canini, Nick Feamster, Jennifer Rexford, and Walter Willinger. 2018. Sonata: Query-Driven Streaming Network Telemetry. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '18)*. 357–371.

[30] Kevin Hsieh, Aaron Harlap, Nandita Vijaykumar, Dimitris Konomis, Gregory R. Ganger, Phillip B. Gibbons, and Onur Mutlu. 2017. Gaia: Geo-Distributed Machine Learning Approaching LAN Speeds. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation* (Boston, MA, USA) *(NSDI'17)*. USENIX Association, USA, 629–647.

[31] Zengfeng Huang, Xuemin Lin, Wenjie Zhang, and Ying Zhang. 2021. Communication-Efficient Distributed Covariance Sketch, with Application to Distributed PCA. *Journal of Machine Learning Research* 22, 80 (2021), 1–38.

[32] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. 2010. ZooKeeper: Wait-Free Coordination for Internet-Scale Systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*.

[33] Ahmad Javaid, Quamar Niyaz, Weiqing Sun, and Mansoor Alam. 2016. A Deep Learning Approach for Network Intrusion Detection System. In *Proceedings of the 9th EAI International Conference on Bio-Inspired Information and Communications Technologies (Formerly BIONETICS)* (New York City, United States) *(BICT'15)*. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), Brussels, BEL, 21–26.

[34] Yiping Kang, Johann Hauswald, Cao Gao, Austin Rovinski, Trevor Mudge, Jason Mars, and Lingjia Tang. 2017. Neurosurgeon: Collaborative Intelligence Between the Cloud and Mobile Edge. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '17)*. 615–629.

[35] Yiping Kang, Johann Hauswald, Cao Gao, Austin Rovinski, Trevor Mudge, Jason Mars, and Lingjia Tang. 2017. Neurosurgeon: Collaborative Intelligence Between the Cloud and Mobile Edge. *SIGARCH Comput. Archit. News* 45, 1 (April 2017), 615–629. https://doi.org/10.1145/3093337.3037698

[36] H Günes Kayacik, A Nur Zincir-Heywood, and Malcolm I Heywood. 2005. Selecting features for intrusion detection: A feature relevance analysis on KDD 99 intrusion detection datasets. In *Proceedings of the third annual conference on privacy, security and trust*, Vol. 94. Citeseer, 1723–1722.

[37] Daniel Keren, Guy Sagy, Amir Abboud, David Ben-David, Assaf Schuster, Izchak Sharfman, and Antonios Deligiannakis. 2014. Geometric Monitoring of Heterogeneous Streams. *IEEE Transactions on Knowledge and Data Engineering* 26, 8 (2014), 1890–1903. https://doi.org/10.1109/TKDE.2013.180

[38] Daniel Keren, Izchak Sharfman, Assaf Schuster, and Avishay Livne. 2012. Shape Sensitive Geometric Monitoring. *IEEE Transactions on Knowledge and Data Engineering* 24, 8 (2012), 1520–1535.

[39] Konstantinos Kloudas, Margarida Mamede, Nuno Preguiça, and Rodrigo Rodrigues. 2015. Pixida: Optimizing Data Parallel Jobs in Wide-Area Data Analytics. *Proc. VLDB Endow.* 9, 2 (Oct. 2015), 72–83.

[40] Arnon Lazerson, Moshe Gabel, Daniel Keren, and Assaf Schuster. 2017. One for All and All for One: Simultaneous Approximation of Multiple Functions over Distributed Streams. In *Proceedings of the 11th ACM International Conference on Distributed and Event-Based Systems (DEBS '17)*. 203–214.

[41] Arnon Lazerson, Daniel Keren, and Assaf Schuster. 2018. Lightweight Monitoring of Distributed Streams. *ACM Trans. Database Syst.* 43, 2, Article 9 (July 2018), 37 pages.

[42] Arnon Lazerson, Izchak Sharfman, Daniel Keren, Assaf Schuster, Minos Garofalakis, and Vasilis Samoladas. 2015. Monitoring Distributed Streams using Convex Decompositions. *Proceedings of the VLDB Endowment* 8 (01 2015), 545–556.

[43] Yi Li, Huy L. Nguyen, and David P. Woodruff. 2014. Turnstile Streaming Algorithms Might as Well Be Linear Sketches. In *Proceedings of the Forty-Sixth Annual ACM Symposium on Theory of Computing (STOC '14)*. 174–183.

[44] Zaoxing Liu, Ran Ben-Basat, Gil Einziger, Yaron Kassner, Vladimir Braverman, Roy Friedman, and Vyas Sekar. 2019. Nitrosketch: Robust and General Sketch-Based Monitoring in Software Switches. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM '19)*. 334–350.

[45] Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, and Vladimir Braverman. 2016. One Sketch to Rule Them All: Rethinking Network Flow Monitoring with UnivMon. In *Proceedings of the 2016 ACM SIGCOMM Conference (SIGCOMM '16)*. 101–114.

[46] Zaoxing Liu, Hun Namkung, Anup Agarwal, Antonis Manousis, Peter Steenkiste, Srinivasan Seshan, and Vyas Sekar. 2021. Sketchy With a Chance of Adoption: Can Sketch-Based Telemetry Be Ready for Prime Time?. In *2021 IEEE 7th International Conference on Network Softwarization (NetSoft)*. IEEE, 9–16.

[47] M. Mönnigmann. 2008. Efficient Calculation of Bounds on Spectra of Hessian Matrices. *SIAM J. Sci. Comput.* 30, 5 (July 2008), 2340–2357.

[48] Seyed Hossein Mortazavi, Mohammad Salehe, Moshe Gabel, and Eyal de Lara. 2020. Feather: Hierarchical Querying for the Edge. In *2020 IEEE/ACM Symposium on Edge Computing (SEC)*. 271–284.

[49] Faisal Nawab, Divyakant Agrawal, and Amr El Abbadi. 2018. DPaxos: Managing Data Closer to Users for Low-Latency and Mobile Applications. In *Proceedings of the 2018 International Conference on Management of Data*. 1221–1236.

[50] Dimitrios Nerantzis and Claire S. Adjiman. 2017. An interval-matrix branch-and-bound algorithm for bounding eigenvalues. *Optimization Methods and Software* 32, 4 (2017), 872–891.

[51] Odysseas Papapetrou and Minos Garofalakis. 2014. Continuous fragmented skylines over distributed streams. In *2014 IEEE 30th International Conference on Data Engineering*. 124–135.

[52] Qifan Pu, Ganesh Ananthanarayanan, Peter Bodik, Srikanth Kandula, Aditya Akella, Paramvir Bahl, and Ion Stoica. 2015. Low Latency Geo-Distributed Data Analytics. *SIGCOMM Comput. Commun. Rev.* 45, 4 (Aug. 2015), 421–434.

[53] T. P. Raptis and A. Passarella. 2017. A distributed data management scheme for industrial IoT environments. In *2017 IEEE 13th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)*. 196–203.

[54] Florin Rusu and Alin Dobra. 2007. Statistical Analysis of Sketch Estimators. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data (SIGMOD '07)*. 187–198.

[55] Vasilis Samoladas and Minos N Garofalakis. 2019. Functional Geometric Monitoring for Distributed Streams. In *EDBT*. 85–96.

[56] Izchak Sharfman, Assaf Schuster, and Daniel Keren. 2007. A Geometric Approach to Monitoring Threshold Functions over Distributed Data Streams. *ACM Trans. Database Syst.* 32 (11 2007).

[57] Izchak Sharfman, Assaf Schuster, and Daniel Keren. 2008. Shape Sensitive Geometric Monitoring. In *Proceedings of the Twenty-Seventh ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS '08)*. 301–310.

[58] Hadar Sivan, Moshe Gabel, and Assaf Schuster. 2020. Online Linear Models for Edge Computing. In *Machine Learning and Knowledge Discovery in Databases (ECML PKDD 2019)*. 645–661.

[59] Hadar Sivan, Moshe Gabel, and Assaf Schuster. 2021. Incremental Sensitivity Analysis for Kernelized Models. In *Machine Learning and Knowledge Discovery in Databases (ECML PKDD 2020)*. 383–398.

[60] Ingo Steinwart and Andreas Christmann. 2008. *Support vector machines*. Springer Science.

[61] C. Stylianopoulos, M. Almgren, O. Landsiedel, and M. Papatriantafilou. 2018. Geometric Monitoring in Action: a Systems Perspective for the Internet of Things. In *2018 IEEE 43rd Conference on Local Computer Networks (LCN)*. 433–436.

[62] Abhishek Tiwari, Brian Ramprasad, Seyed Hossein Mortazavi, Moshe Gabel, and Eyal de Lara. 2019. Reconfigurable Streaming for the Mobile Edge. In *Proceedings of the 20th International Workshop on Mobile Computing Systems and Applications* (Santa Cruz, CA, USA) *(HotMobile '19)*. Association for Computing Machinery, 153–158.

[63] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. 2020. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods* 17 (2020), 261–272.

[64] Ashish Vulimiri, Carlo Curino, Philip Brighten Godfrey, Thomas Jungblut, Konstantinos Karanasos, Jitendra Padhye, and George Varghese. 2015. WANalytics: Geo-Distributed Analytics for a Data Intensive World. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia) *(SIGMOD '15)*. Association for Computing Machinery, New York, NY, USA, 1087–1092.

[65] David P. Woodruff and Qin Zhang. 2012. Tight Bounds for Distributed Functional Monitoring. In *Proceedings of the Forty-Fourth Annual ACM Symposium on Theory of Computing (STOC '12)*. 941–960.

[66] Youngseok Yang, Jeongyoon Eo, Geon-Woo Kim, Joo Yeon Kim, Sanha Lee, Jangho Seo, Won Wook Song, and Byung-Gon Chun. 2019. Apache Nemo: A Framework for Building Distributed Dataflow Optimization Policies. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. 177–190.

[67] Gal Yehuda, Daniel Keren, and Islam Akaria. 2017. Monitoring Properties of Large, Distributed, Dynamic Graphs. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2–11.

[68] Chuanlong Yin, Yuefei Zhu, Jinlong Fei, and Xinzheng He. 2017. A Deep Learning Approach for Intrusion Detection Using Recurrent Neural Networks. *IEEE Access* 5 (2017), 21954–21961.

[69] Minlan Yu. 2019. Network Telemetry: Towards a Top-down Approach. *SIGCOMM Comput. Commun. Rev.* 49, 1 (Feb. 2019), 11–17.

[70] Minlan Yu, Lavanya Jose, and Rui Miao. 2013. Software Defined Traffic Measurement with OpenSketch. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation (nsdi'13)*. 29–42.

[71] Matei Zaharia, Reynold S Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J Franklin, et al. 2016. Apache Spark: a unified engine for big data processing. *Commun. ACM* 59, 11 (2016), 56–65.

[72] S. Zhang, B. Guo, A. Dong, J. He, Z. Xu, and S.X. Chen. 2017. Cautionary Tales on Air-Quality Improvement in Beijing. In *Proceedings of the Royal Society A*, Vol. 472.